Usable Key Management Part 2

For Activity 1, it took me significantly longer to make the key pair signature work than for my macbook, where it took me approximately 3 hours to get the commit to actually be verified in github. My steppers were ultimately all over the place, but it roughly goes like this: Generate a new keypair, add the public key to github, push the commit, notice that it was unverified, notice that something was wrong with the author section machine side, make a new keypad with correct author, try past steps again but now it is not letting me to switch they default key used, finally get that working and still realise that it was unverified, finally fix the real issue by setting the username and password for the hydra machine to match that on github, commit the files and push to see success!

For the most part i did not need any tools for this activity, because I had already done this process before, there were times when i referenced the gpg man pages here https://www.gnupg.org/documentation/manuals/gnupg26/gpg.1.html, to make sure i was using the correct arguments. It is very helpful too and super comprehensive in its use of GPG. And in the midst of my biggest struggle, not knowing why my default key was not changing to the new one I made with the correct author, I asked chatGPT to see what it thought. It was able to correlate guide me out of that struggle so that was a help.

I chose to make a new keypair for this assignment, as that is the more secure option. It is always best to delegate power in situations like this, and splitting up the keys among devices i believe is a more secure way to refer back to key usages The easiest step in my opinion is doing the command that actually creates the key pair, as it guides you though all the different options easily and at the end automatically stores your private key and gives you your public key. The hardest step in this process, at least for me, is actually making sure that the commit is verified, as

there can be a whole number of reasons as to why the commit is showing unverified. Even though it was more difficult for me to do this than last time, i would not change anything about the process because when you figure out how to do it, it is actually quite easy.

ASQ Questions:

- 5
- 2
- 7

Activity 2:

There are a few problematic commits that i see. One of them says that is from the year 2052, and even though its signed correctly that is something certainly to be concerned about. There were also a handful of unverified commits as well as commits that had no signature at all, both of which are fishy but not a total red flag. Another silly problematic commit was the one that said that it was written by copilot, you definitely cannot trust that one as it writes insecure code.

This activity was super quick, there were little to no roadblocks, as the log command within git is very helpful to check configs. There was a time when i was trying to make asc files with the given thumbpritns but i realized i needed the whole keys for those so i just had to manually check. To do this activity, i cloned the repo into my computer, ran a command to see the logs, and then ran another log command that shows the person who signed it as well as their thumbprint. I did not need any tools for this assignment, as it was pretty easy to figure it out on my own amd i already had prior knowledge on how to view logs from my software engineering course.

I cannot distinguish one easy step from the next, as i would say that all steps were equally straight foreard. The one step that i would have considerd difficult would be saving the given public thumbprints and using them to verify the commits automatically. But because i did not need to do that for the completion of the assignment, i did not finish that.

ASQ Questions:

- 7
- 7
- 7

My primary device that i used for commits, my macbook, where i have downloaded GPG Keychain. This stores all of my private keys, so I do not need to worry about them. I was able to continue signing my commits throughout the semester the same way the whole time, It was super easy to use and required no changes to make it continue to work. With how easy it is to sign documents, I could definitely continue to see myself signing all of my commits from now on. It does not hurt at all to sign them, and it is a good practice to continue to grow my computer science knowledge.

SUS questions:

- 2
- 5
- 1
- 5
- 1

- 5
- 1
- 5
- 1
- 5

Github makes it very easy to upload new public keys, and manage them when you begin to have many. I would say overall the hardest part about git signing is making sure all of the details of the signing are correct so that the commit actually shows “verified”, yet it is still not all that difficult. Always the hardest part about git signing is the setup. You do not have to do it often, but because of that when you do have to set it up, it can be a challenge to make sure that you get all of the details in line. So the difficulty goes up and down, with every up in difficulty marked by having to set up the signing on a new device. But overall, it has been a good learning experience and I will continue to use what I learned in this process to continue to sign my commits and be intentional with security in github.

Section 4:

- Are there any security benefits or drawbacks to using Git commit signing as compared to unsigned commits? If so, what are they?
    - Well, the commits that are done in Github are already secure, no change can be made to them after the commit was committed. But the difference that signing brings in is that you can be absolutely sure who wrote or approved the change. Without signing, anyone can say that they are Finn Sparks, and give themselves a

realistic looking email and say that they are me. But if i have a key pair, with a verified and signed commit, then you know that me, the holder of the private key, wrote that commit and no one else could have done it. So There is still a baseline security in git commits, but signing them just gives that extra layer of security.

- To use Git commit signing on multiple devices, you could either synchronize your existing signing key to the new machine or generate a new signing key for that machine. Each of these approaches has different potential security implications. What do you think the security benefits or drawbacks of each approach are?
    - Well it is probably easier to just have one key across all your devices. You do not need to worry about different passphrases or managing multiple keys, but there are more bad things than good when it comes to sharing keys. Your private keys are super valuable, with them come proof of your identity. And by using the same key across multiple devices, an attacker has more ways to imperinate you without you realizing, because one of your devices may be compromised. Another issue is the transfer of keys. If you have any sort of malware or insecurity in your system and you transfer a private key as plaintext, an attacker can get hold of that easier as you bring that key to the light.
- If you were to lose your Git signing private key, what would you need to do to continue signing commits? Are there any security concerns with your proposed workflow? If so, how could they be addressed (you can answer that you don't know)?
    - Well, if you lose the private key, you certainly cannot continue to use it to sign commits. I would recommend removing the public key from your trusted keys, because you do not want a public key accepting a private key that you no longer

have, it could have been stolen. From there you would just have to accept your loss and generate a new key pair.

- If you were to have your Git signing private key stolen, what would you need to do to ensure the security of your repository? Are there any security concerns with your proposed workflow? If so, how could they be addressed (you can answer that you don't know)?
    - This is similar to the last question, all you would need to do is delete the public key from your trust, and then it would be flagged as unverified.
- Consider an open source project that's widely used, and attackers would like to get their code into this repository. What might attackers try to do to compromise that repository? What practices would you recommend that developers of this repo take to ensure that no malicious commits are added to their codebase?
    - If it were me in charge of this large repository, i would have a small “board” of people i really trust. No commit would go though to the actual repo without first getting approval from one of the board members. All of the board members MUST have verified commits so that we can be sure that they are the ones responsible for the changes. This is not a perfect solution, you are banking on the members of the board to be able to check the code for any malicious intent. But this is better than letting anyone and everyone make changes to the repo. To add more security, you could have multiple board members look over the code first before pushing it to main.